

Operating System Fundamentals
Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture – 31
Process Synchronization (Contd.)

So, in our last class we were discussing about solution strategies for the critical section problem.

(Refer Slide Time: 00:28)

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes

The slide also features a small video inset of Prof. Santanu Chattopadhyay in the bottom right corner and logos of IIT Kharagpur and the Department of Electronics and Electrical Communication Engineering in the bottom left corner.

And we have seen three properties that must be satisfied by any solution to the critical section problems. First one is the mutual exclusion that is two processes should not access their critical section simultaneously. So, it must be ensured that whenever a process P_i is executing in its critical section no other process is allowed to enter in their critical sections. So, that is the mutual exclusion condition then there is a progress condition, it says that if no process is executing in its critical section and we have got a number of processes that wish to enter into critical section.

So, right now there is nobody within the critical section, but we have got a number of intending processes who want to enter into the critical section. Then the selection like who will enter into the critical section, it cannot be postponed indefinitely. So, it cannot happen that the decision like who will get chance next, who will be postponed indefinitely so, that is the progress condition. So, decision should be within a finite

amount of time. Then bounded waiting, bounded waiting tells that there must be a bound on the number of times other processes allowed to enter into their critical sections, after a process has made request to enter into critical section.

So, once the process has arrived and it has expressed its intention that I want to enter into the critical section there must be a bound on the number of times the process may have to wait, before for some other processes before it gets into the critical section. A typical situation that I was explaining in the last class is that a ticketing counter and there may be a queue. So, whenever we join at the end of the queue we can just count like how many people are there in front of me. And, then we can get an idea like what will be the approximate timing for getting into the ticket counter.

So, I if every person takes fixed amount of time; so, that time multiplied by number of persons will give me the time for reaching the counter or entering into the critical section. Now, if we assume that there are other sources like there are some mobile calls coming to the ticket vendor and then the person is attending to those calls also. And, there is no bound on the number of mobile calls that the vendor will attend to before serving the next customer in the counter. So, then this bounded waiting condition is not satisfied.

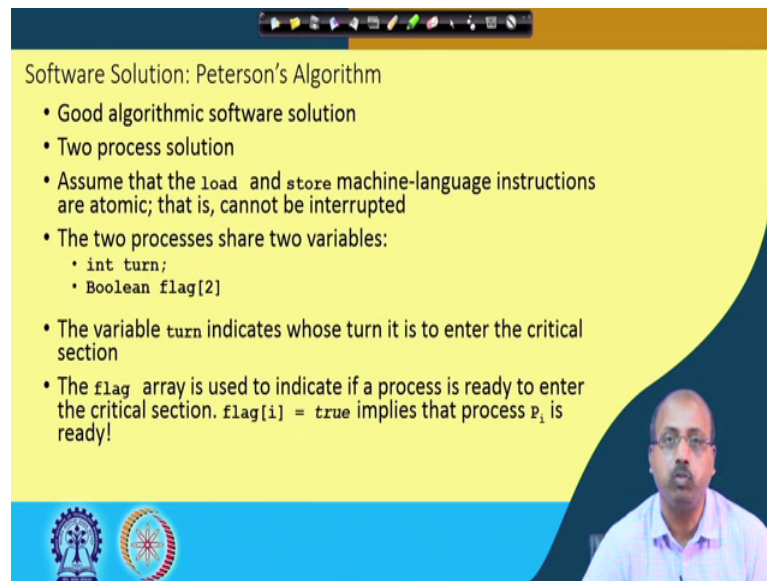
So, these are the three things that are very important. So, any solution that you propose the critical section problem it must satisfy these three conditions. Now, there are certain assumptions also in bounded to waiting condition that is assume that each process executes as nonzero speed. So, every process has got some delay associated with it. So, there is nothing like the process does not take any time for execution. Secondly there is no assumption concerning relative speed of the n processes. So, some of the processes may be very slow, some of the processes may be very fast that way.

Maybe for the same type of statement two processes may take different amount of time because, it is very much dependent on the resources available in the ways. For example, when a process both the processes may be accessing some memory locations. But, in one case the memory location that the process is trying to end access is within is present in the physical memory. In the other case it is not present in the physical memory it is actually a hard disk location. So, that way when you go to the memory management

chapter so, this will be more clear. So, that way the relative speed is not predictable. So, there can be arbitrary relative speed between the processes.

So, whichever solution we propose for the mutual section, for the critical section problem, we must ensure that all these three conditions are satisfied.

(Refer Slide Time: 01:15)



Software Solution: Peterson's Algorithm

- Good algorithmic software solution
- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
 - `int turn;`
 - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process p_i is ready!

The slide also features a video inset of a man speaking in the bottom right corner and two logos at the bottom left.

The first solution that we look into is a software solution known as Peterson solution. So, this is a good algorithmic software solution. So, this is a pure algorithmic solution that we look into and it is a two process solution. So, it works only for two concurrent processes, not for more than two. We assume that the load and store machine language instructions are atomic that is they cannot be interrupted. So, we have discussed about atomic operations in some classes earlier. So, what it means is that when a processor is executing a single instruction interrupt can come, but the current executing instruction is completed first then only it goes into the interrupt service routine.

So, this load and store type of instructions that we will have in the Peterson's algorithm so, they will be assume to be atomic so, we cannot it; they cannot be interrupted. So, it uses two variables: one variable is an integer variable called `turn` which can take the value say 1 and 2 `turn` equal to 1 is for the first process, `turn` equal to 2 means is the second process and there is another Boolean flag. So, that is the again two locations `flag` 0 and `flag` 1. So, whenever the variable `turn` indicates whose turn it is to enter into the critical section so, it goes by a turn.

So, if there is a contention between two processes to enter into the critical section the turn variable will resolve the conflict. So, it will decide whatever be the value of turn that particular process will be allowed to enter into the critical section. And, by this flag variable it is used to indicate if a process is ready to enter into the critical section. So, when a process wants to enter into the critical section it will raise its flag, it will check the flag of the other variable. And, then if it finds that variable that flag is also high then it will be we need to consult the turn variable to decide like which process be allowed to enter into the critical section.

So, flag i equal to true it implies that process P i is ready to enter into the critical section. So, we have got two variables, one is an integer variable turn and an array of two Boolean variables flag.

(Refer Slide Time: 06:34)

Algorithm for Process P_i, P₂

```

do {
    flag[i] = true;
    turn = i;
    while (flag[j] && turn == j);
    critical section
    flag[i] = false;
    remainder section
} while (true);

```

Handwritten notes on the slide:

- i, j with arrows pointing to the indices in the code.
- $\{1, 2\}$ in a box.
- $i=1, j=2$
- ~~$i=1, j=2$~~
- $i=2, j=1$

So, the process P i so, i can be equal to 1 or 2 and i and j so, these are two indices that we will be using. So, here I; so, we have got i and j and so, the i the variable i can take up any value between 1 and 2; similarly j can also take a value between 1 and 2 and they are complementary. So, if i equal to 1 in that case j is equal to 2 and vice versa, if j equal to 1 then i is equal to or rather I should write if i is equal to 2 then j is equal to 1. So, they are complementary to each other in this description of the algorithm. So, if you take i equal to 1 so, I am describing the process P 1, similarly the same process structure will go if I take i equal to 2 that is and that case I am writing the code for process P 2.

So, there are two so; there are codes for two processes P 1 and P 2 as far as this critical section entry exit is concerned so, they are exactly similar. So, they are not written separately, but you can understand that there are two processes, one is process P 1 another is process P 2. So, you can write their codes separately. So, while writing for P 1 so, in this particular course; in this particular code I will replace all i's by 1 and j's by 2 ok. I will do it like this and when I am writing for process P 2; so, this is called P 1 and when I am writing for P 2 in that case I will replace all i's by 2 and j's by 1 ok. So, that is the idea; so, let us try to see how these thing works ok.

(Refer Slide Time: 08:25)

The slide displays the following code for Process P_i:

```

do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    // critical section
    flag[i] = false;
    // remainder section
} while (true);

```

Handwritten annotations on the slide include:

- At the top right: $i=1$, $flag[1]=T$, $flag[2]=F$, $turn=2$.
- Below that: P_1 with $flag[1]=T$ and $turn=2$; P_2 with $flag[2]=F$ and $turn=1$.

So, for the process P i so, suppose i equal to 1 for the process P 1. So, it first raises its flag so, I have got the variables flag 1 and flag 2, I have got these two variables and there is a variable turn. So, initially these flags are all false; so, this is also false, this is also false and turn variable may be arbitrarily set to any value. It may be either 1 or 2, arbitrarily suppose the value is equal to 1 arbitrarily. Now, flag i equal to true so, process 1 wants to enter into the critical section, this flag is made equal to true.

And, then it purposefully makes turn equal to j, purposefully makes turn equal to 2. Then it enters into a while loop checking while flag j and turn equal to j. So, if the process 2 has also expressed its desire to enter into the critical section then it has raised its flag also to be equal to 2. Now, this turn equal to j, this particular instruction is a memory modification instruction that store type of instruction. So, basically so this instruction

both the processes have executed, but whichever process has executed last so, based on that this turn has been set to a proper value. So, if I assume that first the process P 2 came and then it set this flag to equal to true and 2 and turn equal to 1.

And, then process 1 came and process 1 set this turn a flag equal to flag 1 equal to true and turn equal to 2. Then when this process 1 comes to this statement it finds that process 2's flag is on and turn is also equal to 2. So, in that case process 2 will be a process 1 will be waiting at this point. So, when process 1 so, process 1 will get scheduled next, it will find that this flag is equal to it will find that flag j equal to 1 like j is true, but turn is equal to 2. So, that way it will; so, it will not be waiting there. So, turn is not acquired turn is for process 2 so, I have got two processes process 1 and process 2.

So, what has happened is that first process 2 came so, accordingly flag 2 is true and it has set turn to be equal to 1 and, now process 1 came. Process 1 it has done what? It has done flag 1 equal to true and then it has set turn to be equal to 2. Now, when process 1 executes this line it finds that flag 2 is equal to true and turn is also equal to 2 true equal to 2. So, the process 1 will be looping at these statements after some time process 1 will get descheduled, process 2 will come into existence. So, process 2 it will find that flag 1 is equal to true, but turn is equal to 2 because this part was executed first then process 1 came. So, final value of turn is equal to 2. So, as a result process 2 it will find that turn is not equal to 1. So, it will be coming out of this loop and it will be executing the critical section.

After executing the critical section it will set the flag to be equal to false. So, after some time it will set the flag to be equal to false and then next time process 1 gets scheduled. So, it will find that flag 2 is equal to false. So, it will come out of this while loop and then it will enter into the critical section. So, what is happening is that simultaneously both the processes are not entering into critical section. So, once a process wants to enter into critical section, it sets its flag to be true. It sets turn to be the turn of the other process, if the other process has also got flag there its flag raised then this process will be waiting ok. But, the other process when it comes so, it will find that turn is equal to itself. So, it will be coming out of the while loop and go into the critical section.

So, that is how this algorithm works ok. So, so we can look into its satisfaction of properties of this mutual exclusion like this.

(Refer Slide Time: 12:59)

Peterson's Solution (Cont.)

- Provable that the three CS requirements are met:
 1. Mutual exclusion is preserved
 - P_i enters CS only if:
 - either `flag[j] = false` or `turn = i`
 2. Progress requirement is satisfied
 3. Bounded-waiting requirement is met
- What about a solution to $N > 2$ processes

So, provable that three critical section requirements are made. First of all mutual exclusion is preserved because P_i will enter into critical section only if either `flag[j]` equal to false or `turn` equal to i . So, if we look into this instruction this portion. So, you see that you can come to this critical section by failing in this while loop j . So, how can this loop j fail? So, either this is equal to false either this is equal to false or this `turn` is equal to i either of these two conditions has been satisfied ok. So, we can say that as it is written here so, this condition this either `flag[j]` is false or `turn` equal to i .

So, this cannot be satisfied simultaneously because `turn` is a memory location and as I said that since it is an integer variable so, it cannot have two values simultaneously. So, `turn` cannot be equal to 1 and 2 simultaneously; as a result for only one of the processes this `turn` variable will be set to true; set this `turn` condition check will be satisfied. So, if both the processes have raised their request to enter into critical section. So, this flag variables will be true, but `turn` can have only one value at a time. So, whichever process has got that thing satisfied so, that will enter into the critical section, whichever process has got the `turn`. So, that will enter into the critical section other one will be waiting.

So, we are solely relying on the property that whenever you are trying to store some value into the memory location `turn` so, that is an atomic operation. So, it is not interruptible so, as a result it cannot change, we cannot have two value simultaneously. So, that is why mutual exclusion is satisfied, progress requirement is satisfied. So,

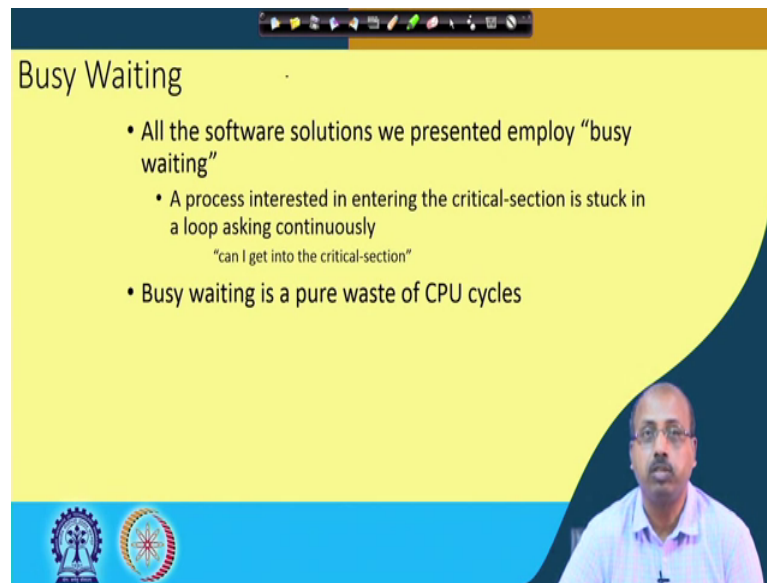
progress requirement says that if you have got a number of processes waiting outside their critical section and they want to enter into the critical section, then the decision like who will enter next so, it cannot be postponed indefinitely.

And, you see that it is decided solely by the turn variable. So, by the turn variable so, it is deciding like which process will enter into the critical section. So, only for one of the processes that while loop condition will be true for the other one it is definitely false. And, whichever process this while loop condition is false, it will enter the critical section then after finishing the critical section it will set this flag to be equal to false. So, you can say that the decision like who will enter into the critical section so, that is never getting postponed indefinitely. So, turn variable is ensuring there.

Bounded waiting requirement is also met because of the situation that the so, after a process has finish the critical section. So, if your process comes and it finds that the other processes in critical section and it cannot enter into the critical section. Then we know that when the other process will finish off the critical section the flag variable will be will reset to false. And, the next time this process will check the while loop condition it will find that flag is the other flag is false so, that way it will be entering into the critical section.

So, bounded wait; it is bounded by at most one execution of the other process. So, bounded wait is also satisfied. So, this Peterson solution it works well for this situations like two process synchronization it works well. What about a solution to N greater than 2 processes? So, there is another algorithm called Bakery algorithm so, we will come to that later after we have discussed about other solutions for this thing. And, so, we will; and there are other solutions to the mutual exclusion problem or the critical section problem using some other strategies.

(Refer Slide Time: 17:05)



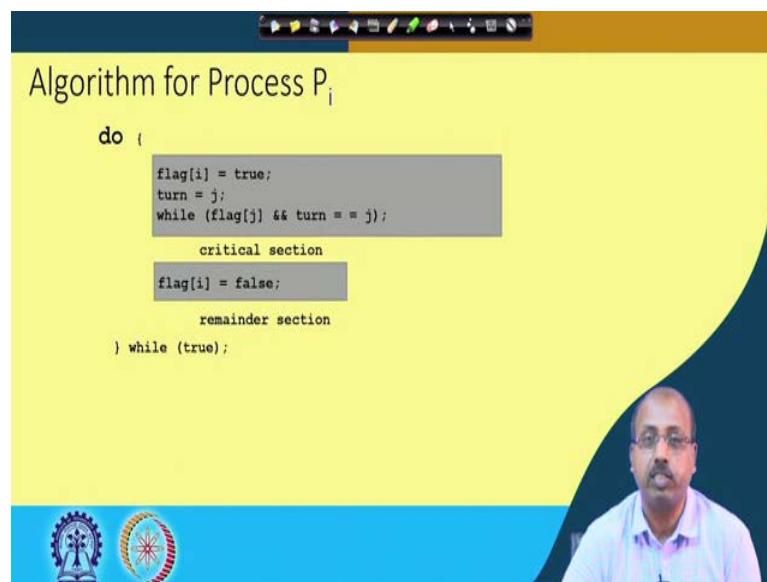
Busy Waiting

- All the software solutions we presented employ “busy waiting”
 - A process interested in entering the critical-section is stuck in a loop asking continuously “can I get into the critical-section”
- Busy waiting is a pure waste of CPU cycles

The slide features a yellow background with a dark blue curved shape on the right. At the bottom left, there are two circular logos. A video inset of a man in a light blue shirt is in the bottom right corner.

So, we will look into those in our next, the point to be noted here is that the software solutions that we have seen so, they employ busy waiting. So, basically that process was waiting in that while loop.

(Refer Slide Time: 17:13)



Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    critical section  
    flag[i] = false;  
    remainder section  
} while (true);
```

The slide features a yellow background with a dark blue curved shape on the right. At the bottom left, there are two circular logos. A video inset of a man in a light blue shirt is in the bottom right corner.

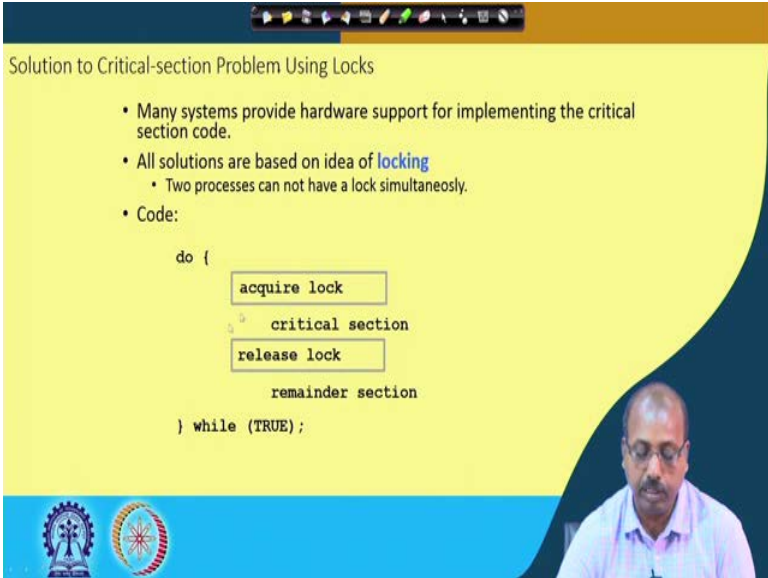
So, in these statement if you see that while flag j and turn equal to j so, this is an infinite loop. So, as far as the processor is concerned so, it is doing something though my process is not advancing it is not it is a looping in that a while statement, but the processor is busy, processor is doing something. So, this type of situation so, they are known as busy

waiting. A process interested in entering in the critical section is stuck in a loop asking continually can I get into the critical section. So, this is the policy in the busy waiting.

And, busy waiting is definitely wastage of the CPU cycle because, there nothing fruitful computation is being done. And, we are simply checking the values of some memory location to see whether they have got the proper value. So, that I can be allowed into the critical section. A better situation is like this that when the situation becomes; condition becomes available then the process may be notified. So, it is typically the situation like if there is some counter where you need to go at some regular intervals or time and see whether something is available or not.

So, otherwise; the other option is that when the item comes then you are informed that the item has arrived so, you can come now. So, this is the difference. So, first one is a busy waiting when you are going again and again to the counter checking for the item. So, that is the busy waiting sort of thing. Other case so, you are just doing something else and when the item arrives then you are informed so, that is a better strategy. So, in operating system implementation also so, we can do some such things as we will see.

(Refer Slide Time: 18:53)



The slide is titled "Solution to Critical-section Problem Using Locks". It contains the following content:

- Many systems provide hardware support for implementing the critical section code.
- All solutions are based on idea of **locking**
 - Two processes can not have a lock simultaneously.
- Code:

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

The slide also features a video inset of a man in the bottom right corner and logos of institutions in the bottom left corner.

So, many systems provide hardware support for implementing the critical section code and all solutions are based on the idea of locking. So, locking means some lock will be acquired. So, if a lock is set since the other process will not be able to acquire the lock and to enter into the critical section; so, it is necessary that the process acquires the locks.

So, two processes cannot have lock simultaneously. So, the code is like this, it is again an infinite loop do while loop so, acquire lock.

So, process which is entering or trying to enter into the critical section, it tries to acquire the lock. Then if the lock has been acquired properly, then the process will enter into the critical section and after finishing the critical section it will release the lock, then the remainder of the same remainder section. So, this way I can have a system solution to the critical section problem; however, here the lock is provided by the hardware.

So, as I was telling in our introductory classes that these operating system designers and processor designers they have to work hand in hand for having some fruitful facilities created for the OS design. So, this is one such facility that the hardware designer should provide, should provide some sort of locking mechanism. When that lock can be acquired by executing some statement and lock can be released by executing some statement. So, if the lock has been acquired properly then we can check whether the lock has been set or not and then enter into the critical section. With the idea that until and unless the lock is released by this process no other process will be able to acquire the lock.

(Refer Slide Time: 20:38)

Synchronization Hardware

- Modern machines provide special atomic hardware instructions to implement locks
 - **Atomic** = non-interruptible
- Two types instructions:
 - Test memory word and set value
 - Swap contents of two memory words

lock: 0

Check lock value. If reset, set it to 1 and enter CS

mem read

mem write

LOAD lock, R

STORE lock, R

Synchronization hardware so, modern machines they provide special atomic hardware instructions to implement the locks so, lock is a concept. So, how these locks are provided? So, they are by means of some atomic instructions. So, atomic means as I said

that they are non-interruptible. So, you cannot divide an instruction to stop in between so, that is not possible. So, and if any machine instruction so, that is going to be an atomic instruction. There are two types of instructions that we will be looking into test memory word and set value. So, they are also known as test and set type of instruction.

So, in a single instruction so, we will test the content of a memory location and save the value; you see that if this was not there. So, I can have a memory location whose name is say lock whose name is say lock. So, I can initially reset the lock to 0 and when to set the lock value I first need to check the lock value. So, it is like this I want to do this thing check lock value, if reset then set it to 1 and enter critical section. So, this is the whole thing that I want by a lock.

Now, if so, you see there are two memory operations that is needed: one is check the lock value. So, this check lock value so, this can be since it is a memory location so, this is a memory read operation. This is a memory read operation then after that if it is reset then I want to set it to 1. So, set it to 1 so, this is a memory write operation. So, we have got two operations on the memory for executing this whole thing. So, there should be a memory read operation and followed by a memory write operation. And, the problem is that this memory read memory write type of instructions that are available in a processor so, they are atomic.

But, these two statements they are not atomic altogether so, this may be a memory read. So, which is basically a load instruction and this memory write is a store instruction. So, I can say like load lock so, as a result I will get the value of lock memory location into some register R and I can say store lock R. So, then by this I set; before this I set the some R to be equal to say 1 and then I store lock R. So, that will be storing the value of R into lock so, lock will be set to 1.

So, now you see that these three instructions that I have written so, all these three instructions. So, interrupt can come after executing the first instruction, after executing the second instruction etcetera. So, altogether these three instruction block is not atomic in nature ok. So, what this test memory word and set value; so, this instruction is going to provide us is it provides these three instruction facility together. In a single instruction these three instructions are clubbed, as a result this processor cannot be interrupted in

between ok. So, you cannot have this atomicity lost. So, this entire thing is executed atomically. So, that is the test memory word and set value type of instruction.

So, otherwise it is nothing, but a memory read followed by memory write, but the difference is that this whole thing is done atomically. So, processor cannot be interrupted in between these read, write operations. Other instruction that we have is to swap the contents of two memory locations. So, I can have a variable I can have a memory location called lock, I can have another memory location called key. So, I can just swap the content of lock and key to get the previous value of lock into key and I can previously set key to 1. So, that after executing this swap instruction the lock variable becomes equal to 1. So, that way I can set the lock as well as check the value of the previous lock value, previous lock location by a single instruction.

So, this is again nothing, but some memory read write operations that I am trying to do, but the point is that all of them are executed atomically. So, we cannot stop in between. So, this both of these two instructions so, they are going to be atomic in nature. So, this is what this processor designer they have to implement these type of instructions. And, if these instructions are available then in the design of the operating system we can take help of these features.

(Refer Slide Time: 25:51)

test_and_set Instruction

- Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Handwritten annotations: 'target' with an arrow pointing to the parameter, 'FALSE' with an arrow pointing to the return value.
- Properties:
 - Executed atomically
 - Returns the original value of passed parameter
 - Set the new value of passed parameter to "TRUE"

So, this is the test and set instruction definition. So, as I have said that this is just a pseudo code that we have here you. So, apparently it seems that this is nothing, but a

piece of program that can be executed by any processor. But, the good thing is that this whole thing is atomic in nature, this whole execution is atomic in nature. So, it is executed atomically. So, what this instruction is doing? So, it is taking the value of the; your target location is passed as an the address of the target location is passed. Then the from that location the content is taken into a temporary variable rv, then this target is set to true and it returns the value of rv.

So, if this is the location target, if this is the location target then initially that location maybe having false ok. So, after executing this piece of code atomically what will happen? So, this location will be set to true and the return value for this function will be false. So, this particular feature it can be executed, it can be utilized for implementing this solution to the critical section problem. It returns the original value of passed parameter and it set the new value of passed parameter to true. So, this is the test and set instruction meaning ok.

So, this processor designer they have to implement it in such a fashion that this particular semantics is implemented in the processor in the instruction.

(Refer Slide Time: 27:37)

The slide is titled "Solution using test_and_set()". It contains the following text and code:

- Shared Boolean variable `lock`, initialized to FALSE
- Each process, wishing to execute critical-section code:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
        /* critical section */  
    lock = false;  
        /* remainder section */  
} while (true);
```

Handwritten notes on the slide include:

- A box labeled "lock" containing "PZE".
- Labels P_1 , P_2 , and P_3 with arrows pointing to the `test_and_set(&lock)` line in the code.
- Three circles labeled P_2 , P_1 , and P_3 drawn below the code.

At the bottom of the slide, there are two bullet points:

- What about bounded waiting? ✗
- Solution results in busy waiting.

The slide also features a video feed of a man in the bottom right corner and a Windows taskbar at the bottom.

So, we will see this what how can I use it for solving this critical section problem, it is like this. So, we have got a shared Boolean variable lock initialized to false, suppose I have got a number of processes say P 1 P 2 P 3 etcetera. Now, every process structure is like this. So, all the processes their structure is same as this one. So, I have got a shared

location whose name is lock; whose name is lock and this lock is initialized to false ok. So, we have got a shared location initialized to false. Each process wishing to execute the critical section code so, it will do like this.

So, while test and set lock do nothing. So, it will try to; so, the first process set P 1 comes, P 1 executes this test and set instruction. So, as a result this is set to true and it gets the return value of this test and set as false fine. So, for process P 1 there is no barrier now so, it enters into the critical section. Now, when this process P 1 is in critical section suppose P 2 also comes and it enter tries to enter into the critical section. So, it will again execute a test and set instruction, but this time P 2 gets the previous value up lock which is equal to true. So, P 2 gets it as true so, P 2 will be waiting here. So, if P 3 comes so, P 3 also executes a similar instruction and P 3 will also be waiting there. So, it will be getting the value true.

So, as a result only P 1 is in critical section, P 2 P 3 they are not they could not enter into the critical section. Now, after some time P 1 will finish off so P 1 will set this lock to be equal to false again. So, next time P 2 executes this while loop condition check. So, it will get this test and set return value as false. So P 2 will enter into the critical section or if P 3 had come before P 2 then P 3 will do this check and it will find that return value is false. So P 3 will enter into the critical section.

So, as a result you see this solution that we have so, mutual exclusion is definitely satisfied. Bounded waiting so, bounded waiting cannot be guaranteed because we do not know like once a process P 2 has ask to enter into the critical section, we do not know when will it be allowed because it may. So, happen that scheduler picks up the processes P 3 and P 1 always it so, every time P 2 is in P 2 this P P 1 was in critical section. So, after finishing P 1 the scheduler may schedule P 3 so, P 3 is execute.

So, P 2 comes in between P 2 finds that the lock is not available, it again waits then after some time P 3 finishes again maybe by this time P 1 has also arrived. So, P 1 and P 2 both are waiting so, scheduler picks up P 1. So, as a result P 2 again wait. So, there is no limit, there is no bound on the number of times P 2 has to wait before getting into the critical section. So, bounded weight is not satisfied so, bounded waiting is not satisfied in this solution. And, this solution is definitely a busy wait solution because it is continually

checking this, executing this test and set instruction. And, it is trying to see whether this permission is granted for going into the critical section.

So, we will continue with this in the next class.